HAL open science

Wcomp: a Multi-Design Approach for Prototyping Applications using Heterogeneous Resources

Daniel Cheung-Foo-Wo, Jean-Yves Tigli, Stéphane Lavirotte, Michel Riveill

To cite this version:

Daniel Cheung-Foo-Wo, Jean-Yves Tigli, Stéphane Lavirotte, Michel Riveill. Wcomp: a Multi-Design Approach for Prototyping Applications using Heterogeneous Resources. 17th IEEE International Workshop on Rapid System Prototyping, IEEE Computer Society, Jun 2006, Chania, Greece. hal-00481796

HAL Id: hal-00481796

https://hal.science/hal-00481796

Submitted on 7 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

Wcomp: a Multi-Design Approach for Prototyping Applications using Heterogeneous Resources

Daniel Cheung
CSTB
290, route des Lucioles, BP209
06904 Sophia-Antipolis, France.
daniel.cheung@cstb.fr

Jean-Yves Tigli, Stéphane Lavirotte, Michel Riveill
I3S Laboratory – (UNSA / CNRS)
Bât. ESSI - 930 route des Colles, BP 145
06903 Sophia-Antipolis cedex, France.
tigli@essi.fr, lavirott@unice.fr, riveill@essi.fr

Abstract

This paper presents Wcomp which is a framework for rapid application prototyping. This framework has been developed for targeting wearable computing applications but can also be used in the field of pervasive and context-aware computing. In the first part of the paper, we investigate the possibility of taking into consideration the relations between software components and resources of the "operating context" in our Wcomp platform. Secondly, we investigate the opportunity of taking a multi-designer approach in order to adapt the application to multiple well-suited representations. Then we introduce in the platform a new design approach based on patterns of interactions called ISL4Wcomp.

1 Introduction

Computing environments are not composed of standardized entities such as standard PC computers or the standard WIMP (standing for "Window, Icon, Menu, Pointing device") human-computer interaction anymore. WIMP is based on an encapsulated graphical runtime and the traditional trio: display, keyboard and mouse. It is a standardized multi-task operating system often reinforced by a standard virtual machine layer and its associated framework. For pervasive and ad-hoc computing, we cannot in fact design applications if neglecting the various operating contexts. We need to explicitly and clearly manage the dependencies between components and subsystems.

For *Operating and Embedded systems*, in [1], [2] and [3], the authors studied hardware system resources and their limitations. We gather this work under the label "Software and Hardware System resources and context".

For *Multi User Devices*, in [4] and [5], the authors focus on various I/O physical devices and their management. This could be labeled "Human-Machine Interaction resources and context".

Finally, for *Multi Networked Devices System*, in [6], we refer to the work of Cervantes and Hall on the dynamic reconfiguration of networked devices and associated *services* (or drivers).

Most of these approaches highlight domain specific dependencies between components and software subsystems. But none considers all the subsystems included in the overall operating context. According to the kind of resources they deal with, these systems might be qualified as embedded, embodied or situated. An *embedded* computer has constrains about memory, speed and so on. The purpose of an *embodied* computer interacts continuously with its environment. A *situated* computer belongs to the environment.

So, in order to define a mobile and ubiquitous computing application, we need to deal with different formalisms. The main challenge is that these tools must share data in order to support consistency checking and reuse. The Wcomp platform was first designed to implement prototypes of this kind of applications using a components-based approach. And as soon as a new component appears and the operating context changes, the component assembly should be rebuilt. It can be complex to analyse the global graph of an application. Not only have we needed to ensure the separation of concerns but also the validity of the final result.

1.1 A component-based framework

There exist several definitions for the term *component*. One is given by Clemens Szyperski in [8]: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*" We study in this part how current approaches demonstrate these context dependencies.

In Wcomp approach, we consider that such relations between components and resources are characteristics of the operating context of the application. In component-based approaches, this level can create, add and remove components as well as connectors in order to modify the applica-

tion at the *programming level*. We can distinguish various useful design approaches. Each approach is based on a different representation and manipulation of the components, connectors and their assembly. The main influences of the *design level* on the *executive level* can be structured on three levels of capabilities:

- designing the program (creating components)
- modifying the program (changing components)
- modifying control flows between components

At the same time, we can classify the representations and the manipulations of the components assembly according to their user interface: visual or textual, programming oriented or rendering oriented.

1.2 Integrated Development Environments (IDE)

Today, many integrated development environments adopt various hybrid approaches. For example, as far as software industry is concerned, with Visual Studio IDE, Microsoft mixes both design tools from the visual programming approach as it represents the rendering of the graphical application, and the textual programming approach as it modifies some parts of the source code of the application. Meanwhile, the JavaBeans approach of Sun is a self-sufficient visual design approach because it allows the user to manipulate data and to control flows. It uses an event-based graph in order to design interactions between components [9].

However, those approaches are not without limits; this is mainly the consequence of the heterogeneity of resources which should be considered when writing programs for pervasive or context-aware computers. Consequently, the expected design environment needs to provide various adapted representations and design tools to develop the application.

In this paper, we present Wcomp as a framework for prototyping pervasive, context-aware and wearable computing applications. The first section describes the overall system. We present the *component-based* approach and its ability to manage heterogeneous operating contexts, the diversity of their designs and programming approaches. In the second and third section, we present the Wcomp approach and the recent improvements to deal with a changing operating context, and developing *designers* which we have implemented to address the issues linked with the programming and adaptation of the application. Then, we present an example of the Wcomp application in the field of home automation taking advantage of these recent contributions and presenting the technologies we used. Finally, we conclude with discussing on the limits of our approach and suggest directions for future research.

2 Our rapid prototyping environment: Wcomp

With Wcomp [7], we explore an overall approach consisting of three levels: context level, design level and executive level. The first level is composed of three contextual elements: software components, resources (software subsystems) and specific devices. The second level (the design level) provides various representations and design tools to create, configure and adapt the application based on components. The third level (the executive level) controls the discovery of new contextual elements and adapts the components assembly so that it can deal with the new context.

2.1 Description of the Wcomp framework

First of all, we have to define what we consider as a component in Wcomp. The Wcomp component model is inspired from the JavaBeans model. But it has been slightly modified. A component in Wcomp is still an instance of a class. But it is not necessarily *serializable*. A component has a unique name. We consider $\mathcal{C}$ the set of components. A component has an interface which has two sets composed of events and of methods. We call E the set of events characterized by their unique name and M, the set of methods. Let us gather the definitions of events and method definitions in the term "port". We consider a set of links $\mathcal{L}$. A link is a list composed of an event and of a list of methods. An assembly consists of a subset of $\mathcal{C}$ and $\mathcal{L}$. The *container component* implements an API to control programmatically this assembly. It implements consequently the addition and removal of elements in $\mathcal{C}$ and $\mathcal{L}$.

The *context level* represents the resources and the components we have to deal with during the design of the application. Such resources are often directly and exclusively managed by the operating system in most component-oriented approaches. In Wcomp, we have not yet an explicit model of the resources of the context. Nevertheless, we classify our components according to their implicit interactions with particular resources. For example, we distinguish active or passive components according to their being coupled with a *system thread* resource or not. In the same way, we distinguish mixed (hardware and software) or purely software components according to their being dependent on a physical device or not.

In this way, we increment the Wcomp model via the introduction of the set of resources $\mathcal{R}$. Thus, we introduce in our model *implicit interactions* which are interactions between resources and components and/or resources. Such a representation of resources allows consequently our component model to strictly respect the Szyperski's concept of explicit context dependencies.

2.2 Design models

As we mentioned in the introduction, we believe that a design environment should provide various representations and design tools for programmers to work on an application. We present a multi-designer approach which allows to modify and then adapt applications thanks to the use of *different design views*.

2.2.1 Source-code designer

```
    //
    this.button1.Click += new System.EventHandler(this.__butto
}

private void __button1_to_counter1_0(object sender, System.Eve
    this.counter1.Increment();
}
#endregion
```

Figure 1: Source-code designer

We consider as source-code designer (Figure 1) one particular case (only one target system and one language): the compiled-on-the-fly C# .NET code. This source code represents the application based on components assembly. When the programmer modifies this source code, the *source designer* communicates the modifications to the *container*. Those modifications consist in the replacement of component instances if their source code has been changed. They modify links between components as well.

2.2.2 Visual rendering designer

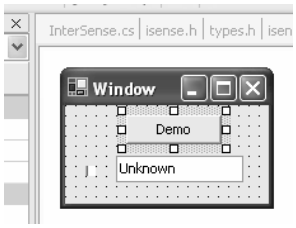

Figure 2: Visual rendering designer

A graphical application has a visual screen rendering. This rendering is manipulated by the programmer via the visual rendering designer. As an example, we propose the design of a graphical application composed of a button, a text field and a checkbox gathered in a window (Figure 2).

2.2.3 Console designer

A console designer (Figure 3) stands for a special designer where we can see an example of sending two *add_component* commands by typing "AddWNBean *Type Name*").

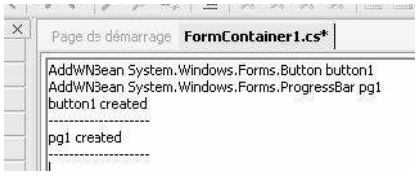

Figure 3: The Console designer

This designer allows to send commands to the container thanks to a simple command language. Each command has a name followed by parameters. There are four *intercessive* commands listed in Table 1. They modify the container contents by respectively adding, linking, removing and unlinking components.

Command
add_component *Type (Name) (X Y)*
link *Source Event Target Method (Params)*
remove_component *Name*
unlink *Source Event Target Method*

Table 1: Intercessive commands

There are introspective commands gathered in Table 2.

Command	Description
list_component_types	Give all available types.
list_components	List all instanciated components.
list_links	List the links.
list_methods *Name*	Give the signature of each method of a comp.
list_events *Name*	Give the signature of each event of a comp.

Table 2: Introspective commands

2.2.4 Graphical component assembly designer

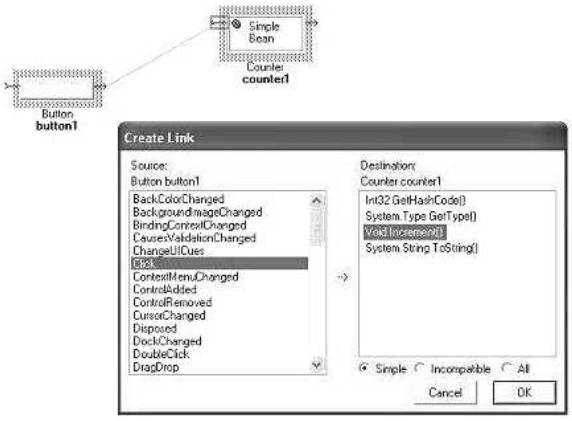

Figure 4: Component assembly designer

The application assembly can be graphically manipulated by the graphical component designer. As an example, we propose the creation of a link. The process of creating a link between two components is shown in Figure 4. To complete this multi-design approach, we propose the study of a new smart designer ISL4Wcomp.

3 A new Wcomp designer: ISL4Wcomp

We explain in this section how we use our previous work on software interactions to build a new particular *designer* for Wcomp framework.

3.1 What is ISL?

The Rainbow team proposes the use of a dedicated **I**nteraction **S**pecification **L**anguage (see [10] and [11]) to express interactions between software components in a component-based application. This approach brings out three major benefits:

- It allows component interactions to be expressed explicitly as first-class entities.
- It enables the expression of interactions independently of any specific language or component model.
- It authorizes the dynamic adaptation of applications as it defines and removes interaction at runtime.

To achieve this, interaction patterns (or simply *interactions*) are specified in ISL. Interactions represent a set of connections between some component instances. An interaction server is in charge of managing the life cycle of interactions such as their registration, their instantiation, their destruction and their merging. *Noah Interaction Server [12]* is the name given to the implementation of the interaction server.

3.2 What is ISL4Wcomp?

To integrate this work into Wcomp model and meet Rapid Application Development purposes, we have adapted ISL language through the definition of a new grammar. The evolution of programming languages sets up new implementations of interactions such as event and delegation concepts in C# language.

Originally ISL language permits to redefine method calls via the relocation of this call at runtime in order to point at a new piece of code which calculates how and when the real method is going to be executed. Meanwhile, Wcomp model uses method calls as inputs of components which can be rewritten by ISL. But it also uses new event constructions as outputs. As a consequence ISL language should have been modified to be able to rewrite those *ports*.

Secondly, ISL language has been created so that two descriptions written in ISL could be composed automatically by the machine. Thus, for the machine to take a decision for all cases it may encounter, the definition of particular operators that we may qualify as compositional-specific has been written. In ISL, those operators are known as *call* and *delegate*. They control the way some parts of patterns are to be composed when in conflict.

More specifically, the keyword *call* is used inside a redefinition of a *port* (it is true for *delegate* as well). It tells that the actions defined by other patterns may appear when a *conflict* occurs (see the following subsection about conflicts). On the contrary, the keyword *delegate* means that the actions that it suggests replace what the other patterns define.

The syntax and the expression of interaction patterns have been homogenized and mimic current high level languages such as C# or Java. This new ISL language construction is called ISL4Wcomp.

3.3 ISL4Wcomp architecture

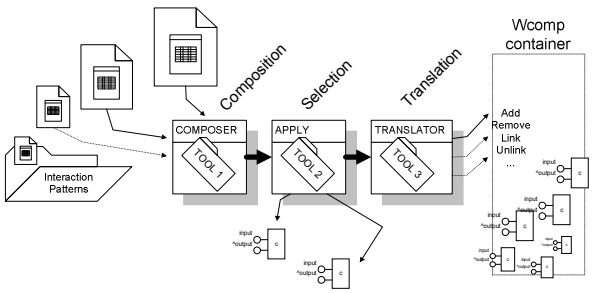

Figure 5: ISL4Wcomp architecture

The architecture of ISL4Wcomp (Figure 5) is different from ISL. We do not reuse the interaction server to manage the life cycle of *interactions* because the language has changed. Instead, we developed a set of separate tools to:

- compose interactions
- apply interactions to component instances
- translate interaction into reassembly commands

3.3.1 Composition

Components can be involved in different interactions that must be composed. A first tool, called *ISLComposer,* achieves this task and takes as inputs interaction patterns and gives, as a result, a single pattern as output. A second tool, called *ApplyISL,* takes a pattern and rewrite it in order its variables to map component instances. Finally, the last tool translates an applied pattern into a set of reassembly commands. It is called the *ISLTranslator*.

ISL4Wcomp enables rapid assembly of component-based applications because it gathers hand-made connections between components into an interaction pattern. Furthermore, it adds *logic* to those connections according to the usage of ISL behavioral keywords such as sequence, parallelism, condition, waiting and signaling. Hence, the programmer builds applications once and translates them into interaction patterns. He can then reuse those patterns as part of another application later. He may also build new applications by instantiating several patterns.

The definition of a pattern library is generally a means to simplify software development. We might go further through the automation of their selection and of their composition. The composition of patterns is calculated by *ISL-Composer*.

The calculation of the composition of interactions requires the definition of (only) twenty-seven rules for combining eight operators : sequence, parallelism, conditional, waiting, signaling, message, call and delegation (see *[12]* for further details). Those rules tell how each operator combines with one another.

3.3.2 Conflict and merging

We have seen that an interaction pattern is structured as a set of twofold rules:

- The first category of rules rewrites method calls.
- The second redefines event emissions.

Those redefinitions are written as if they were bodies of methods. And in these bodies, we describe the behavior replacing the method call or the event emission.

Conflicts occur when two patterns redefine the same method call or event emission. When a conflict occurs, the bodies of conflicting interaction rules are merged.

To illustrate our purposes, we propose the study of a practical example where Wcomp is used to instantiate components and manage relationship between them.

4 Application

The aim of this application is to manage home appliances. We want to illustrate the technologies used to implement this prototype and the advantages to use Wcomp in such a case.

4.1 Operating context and UPnP devices

The component-based application has been designed on the basis of the Universal Plug and Play (UPnP) technology. UPnP is a set of computer network protocols which allow devices to connect seamlessly and to simplify the implementation of networks. The application makes use of a presence detector, a light, a switch and a shutter. UPnP architecture offers pervasive peer-to-peer network connectivity of PCs, home appliances and wireless devices. The devices can be virtual (software simulation) or hardware.

We did not code the explicit integration of the detector in this application. But to do so, the programmer has only to develop the Wcomp component representing the UPnP detector (with the help of a *wizard tool*) and the interactions describing the functionality.

4.2 ISL4Wcomp and house automation application

We study in this section the integration of the detector. First, we see how interactions are composed. Then we study the particular case of merging of rules. And finally, we analyze how the result is integrated into Wcomp.

4.2.1 Pattern composition

Here is the example of the composition of interaction patterns. It consists in modifying a component assembly so that it considers a new functionality such as "opening a shutter rather than turning light on, according to outside luminosity". This assembly was simply defined as such: a detector *d* turns on a light *l* when somebody enters the room and a switch *s* opens a shutter *v* when activated.

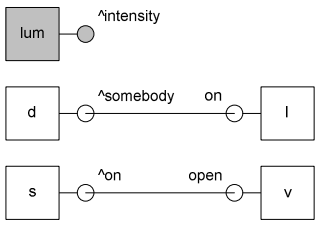

Figure 6: Component assembly in its initial state.

The assembly is translated into the following interaction pattern called *p1*:

```
pattern p1(switch s, light l, shutter v,
           detector d, indicator i) {
  lum.^intensity() { call || i.set()  }
  s.^on()          { call || v.open() }
  d.^somebody()    { call || l.on()   }
}
```

We may notice that each connector is systematically translated into an ISL code "*source* { call ‖ *target* }". This is the first naive algorithm from the moment to translate assembly to *interaction pattern*.

Then we would like to consider a new component into the assembly that is component *lum* which is a detector emitting regularly a value indicating the luminosity outside the house. The new functionality we integrate into the ap-

plication is summarized into the following interaction pattern:

```
pattern p2(switch s, luminosity lum,
           shutter v, comparator c) {
  lum.^intensity(int value) {
    call || comp.set(value);
  }
  l.on() {
    if(c.isEnough())
      delegate { v.open() }
    else
      call
  }
}
```

This interaction pattern describes the two following functionalities. On the one hand, as soon as the luminosity sensor *lum* throws a value describing the intensity, the indicator *i* displays it on a screen. On the other hand, as soon as the switch *s* is activated, the shutter *v* is opened. And as soon as the detector *d* detects that there is somebody in its field, the light *l* in turned on.

Moreover, as soon as the luminosity sensor *lum* throws the light intensity, we *should* tell the comparator *comp* to memorize this value. And instead of turning the light *l* on, we should also check if the luminosity is adequate outside. If it is adequate, nothing has to be done except that the shutter has to be opened. If it is not, we do what other interactions have defined or just turn the light on.

The *application of a pattern* to a set of components consists only in renaming variables used inside an interaction pattern so that they correspond to component instances names defined in the container.

The composition of two patterns implies the homogenization of the parameters of each pattern. The result of the composition of the two sets of parameters is their union. Let's call the resulting pattern *p3=p1+p2*. We have:

```
pattern p3(switch s, luminosity lum,
           shutter v, comparator c, light l,
           detector d, indicator i)
```

The rules that are not conflicting are simply copied to the new interaction. This is the case of the rule *l.on*.

4.2.2 Merging process to solve conflict

When two *interaction rules* are in conflict, they are merged two by two. We may notice that the merging process is commutative. Consequently the order in which the rules are merged does not matter. Rules are composed of a body that can be represented through a *tree syntax*. Each node of this tree represents a keyword. The merging process takes two trees $\mathcal{T}_1$ and $\mathcal{T}_2$. It computes first the root of $\mathcal{T}_1$ and $\mathcal{T}_2$. A rule tells the machine what is the result of the merging of two nodes. Then the merging is called recursively on each leaf of the tree. When the process holds, the remaining tree stands for the behavioral merging of the behaviors of each tree.

We have formalized the process in terms of logical *rewriting-rules*. And for the experimentation, we implement those rules using Prolog language, which has resulted into a shared library.

4.2.3 Translation into command list

We have also written the translator in Prolog. But the latter was rather formalized into set of syntactic analyses of ISL programs which leads to the generation of an adequate list of commands. The analysis requires only one parsing of the ISL code.

```
unlink s ^on v open
add_component IF if0
link s ^on if0 do
add_component COMPARATOR c
link l ^intensity c intensity
link if0 ^cond c greater
link if0 ^then l on
link if0 ^else v open
```

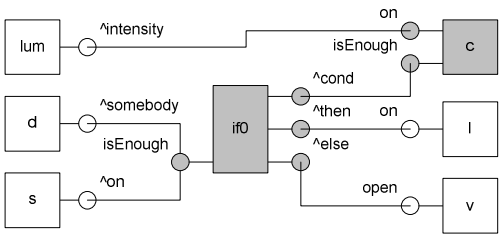

Figure 7: Component assembly in its final state.

4.2.4 Reassembling the application

Each behavioral operator in an ISL program is represented by a component in the *container*. For instance, the conditional operator *if* has a corresponding component in Wcomp which is also called *if* and has one input *void do()* and three outputs *void ^cond()*, *void ^then()* and *void ^else()*.

Those components representing ISL behavioral operators differ from common Wcomp because their inputs and outputs can be connected to any other outputs, respectively inputs. And normally, inputs and outputs are *typed* and can only be connected with corresponding signature. We call those components *generic components*.

A component is qualified as *generic* when its events (outputs) and its method definitions (inputs) can be connected to any other methods or events. Those ports are characterized by their signature. We have creating generic ports by imposing the following signature for a port *p*

object p(object[] data).

The connector linking this port to another port *p'* is responsible for adapting the signature of *p'*, say $r\ p'(a_1,\ldots,a_n)$ to *p*.

Thanks to the *anonymous method* construction introduced in the C# language, we created *on-the-fly* for each connection a first category of methods which saves the arguments $a_1,\ldots,a_n$ into a table of objects called data and a second category of methods which transfers those arguments.

The recognition of signatures is done by the *reflection mechanism* (done only at *design time* not at *execution time* to meet performances).

4.2.5 Undo modifications by removing a pattern

Removal of patterns consists in reversing the semantics and the order following which the commands have been sent to the Wcomp *container*.

```
unlink if0 ^else v open
unlink if0 ^then l on
unlink if0 ^cond c greater
unlink l ^intensity c intensity
remove_component COMPARATOR c
unlink s ^on if0 do
remove_component IF if0
link s ^on v open
```

Finally, the application retrieves its original states. But this algorithm has a main drawback: the last state of the application in terms of components and links should be recorded to reconstruct links that have been removed after the application of the pattern.

5 Conclusions

Component-based frameworks are generally coupled with specific design approaches. In our Wcomp platform, we have studied the possibility of using a multi-designer approach in order to adapt the application to multiple well-suited representations. We presented a new ISL4Wcomp design approach based on interaction patterns. ISL4Wcomp enables rapid application prototyping for applications based on the assembly of components.

Our approach will scale as the size and the complexity of the component-based system grows. We are currently studying the scalability of the techniques and the tools as the complexity of the rules and their number increase. Furthermore, we are working on different evolutions of our platform towards a distributed environment enabling the design of the distributed application to take into account the diversity of the heterogeneous operating context.

Acknowledgments

We would like to thank our colleagues at Rainbow I3S research team (M. Blay-Fornarino, A.-M. Dery-Pinna and D. Emsellem), E. Pascual of CSTB, A. Bourcet and C. Andral students of Polytech' Nice - Sophia Antipolis.

References

[1] Ford B., Back G., Benson G., Lepreau J., Lin A., and Shivers O. The Flux OSKit: a substrate for kernel and language research. In *SIGOPS Oper. Syst. Rev.*, ACM Press, 1997.

[2] Gay D., Levis P., Behren R. V., Welsh M., Brewer E., and Culler D. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation*, 2003.

[3] Müller P., Stich C., Zeidler C. Components @ Work: Component Technology for Embedded Systems. In *27th International Workshop on Component-Based Software Engineering, EUROMICRO*, 2001.

[4] Dragicevic P. and Fekete J.-D. ICON: Input Device Selection and Interaction Configuration. In *Companion proceedings of the 15th ACM symposium on User Interface Software & Technology (UIST'02)*, Paris, France, p. 27-30, 2002.

[5] Greenberg S. and Fitchett C. Phidgets: Easy development of physical interfaces through physical widgets. *In Proceedings of the ACM UIST*, 2001.

[6] Cervantes H. and Hall R. S. Beanome: A Component Model for the OSGi Framework. In *Software Infrastructures for Component-Based Applications on Consumer Devices*, Lausanne, Switzerland, 2002.

[7] Cheung D., Fuchet J, Grillon F., Joulie G. and Tigli J.-Y. Wcomp: Rapid Application Development Toolkit for Wearable computer based on Java. In *Proceedings of IEEE Int. Conference on Systems, Man and Cybernetics*, 2003.

[8] Szyperski C. *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1999.

[9] Sun Microsystems. JavaBeans. http://java.sun.com/products/javabeans

[10] Blay-Fornarino M., Charfi A., Emsellem D., Pinna-Dery A.-M. and Riveill M. Software Interactions, in *Journal of Object Technology*, vol. 3, no. 10, p. 161-180, 2004.

[11] Berger L. *Mise en Oeuvre des Interactions en Environnements Distribues, Compiles et Fortement Types : le Modele MICADO*. Ph. D. Thesis, UNSA - Faculté des sciences et techniques, Ecole doctorale STIC – Informatique, 2001.

[12] Charfi A., Emsellem D., Riveill M. Dynamic component composition in .NET, in *Journal of Object Technology*, vol. 3, no. 2, p. 37-46, Special issue: .NET: The Programmer's Perspective: ECOOP Workshop 2003.